BUSINESS LOGIC SUPPORT

Cross Reference to Copending Applications

The disclosure of this application is related to the disclosures of the following copending applications:

"Text File Interface Support In An Object Oriented Application," serial no. _____, filed _____ (Attorney Docket END9-2000-0080);
"Flexible Help Support In An Object Oriented Application," serial no. _____, filed _____ (Attorney Docket END9-2000-081);
"Dynamic Java Beans For VisualAge For Java," serial no. _____, filed, _____ (Attorney Docket END9-2000-082); and
"Complex Data Navigation, Manipulation And Presentation Support," serial no. _____, filed _____ (Attorney Docket END9-2000-083);
the disclosures of the four above-identified copending applications are hereby incorporated herein by reference in their entireties.

Background Of The Invention

This invention generally relates to supporting business logic in computer applications. More specifically, the invention relates to methods and systems for managing and applying business rules for data used in computer applications.

Many applications include "business logic" as a part of the total function. This is in addition to application logic which is used to retrieve, present and store data; the "business logic" is usually in the form of extra constraints that are imposed on the data (beyond innate data integrity rules) to unsure that it is valid within that business application. The rules for the business logic are (at least from a programming point of view) more arbitrary than the core application logic. For example, a business rule might be that any new orders have to be at least 50 dollars in cost. This in contrast to a base integrity rule that says that the order header must exist before the line items for that order can be added. Business logic also tends to change over time; frequently this is the major cause of updates to the application.

The traditional approach to this has been to write unique code to handle each business verification. Quite often, this code is intermingled with other code that is involved only with data presentation or storage. There are several problems associated to this approach:

1. It is difficult to find out just how a business rule has been implemented. The business logic may be totally mixed up with other processing so that it cannot be easily isolated and modified.

2. Since the business rules are coded in the same classes that are presenting the data, changing the rules means that many parts have to be changed. These changes may

have unintended side effects because the business logic is not sufficiently isolated.

3. This limits the flexibility of the application and makes it hard to maintain.

4. Implementation of a business rule requires programming expertise. It cannot be done using visual programming techniques.

5. Changing a business rule requires changing the application code and reinstalling the application. This can be particularly annoying if the rules change frequently to adapt to changing business conditions.

6. If a rule is implemented in one place in the application, there is no way to verify that it will be invoked in all appropriate circumstances. Some parallel piece of application processing may also need the rule. if it is implemented twice to cover that situation, then there is no assurance that it will be found in both places when it needs to be modified.

Summary Of The Invention

An object of this invention is to provide a mechanism to isolate and externalize the definition of business rules.

Another object of the present invention is to support business rules using visual programming techniques.

A further object of this invention is to programming techniques to set up business rules and that can be used by a business expert who does not need specialized programming skills.

These and other objectives are attained by providing a mechanism to isolate and externalize the definition of business rules, and to support them using visual programming techniques (special editors for Java beans). This means that the rules can be set up by a business expert who does not need specialized programming skills. In addition, the Java beans are implemented as dynamic Java beans (see the above-identified copending application "Dynamic Java Beans for VisualAge for Java."

This means that the business rules can be adjusted by changing their initialization strings on a relational database, so that the rules can be modified without the need to recompile and reinstall the application. They can be altered by an administrator using the custom editors for the beans. Again, no knowledge of Java is needed to set up the rules. The underlying application environment (the Enterprise Application Development Platform - see the above-identified copending application "Complex Data Navigation, Manipulation And Presentation Support") provides a consistent and reliable invocation of the business rules at predicable points of processing. Each rule can specify during which events (add, change, update, delete, etc.) it is to be invoked.

Of course, not all business logic will fall completely into the pattern provided by this invention. Another

problem it solves is the ability to allow code exits (written in Java) to be invoked to call more complicated processing. These exits share in the isolation and reliable invocation of other business logic supported by this invention.

Further benefits and advantages of the invention will become apparent from a consideration of the following detailed description, given with reference to the accompanying drawings, which specify and show preferred embodiments of the invention.

Brief Description Of The Drawings

Figure 1 is a flow chart outlining a method embodying this invention.

Figure 2 is a flow chart illustrating a method embodying a second aspect of this invention.

Figure 3 shows the major elements of a preferred embodiment of the invention.

Detailed Description Of The Preferred Embodiments

With reference to Figure 1, a first aspect of this invention relates to a method for applying business rules to data. The method comprises the steps of, for each rule, invoking the rule at the occurrence of each of a defined set of operations on the data, and identifying a predetermined trigger condition for the rule. The rule

is then processed if the predetermined trigger condition is satisfied.

As shown in Figure 2, a second aspect of the invention relates to a method for managing business rules for data. This method includes the steps of establishing a set of business rules, where each of the rules includes a trigger condition and a process, and storing the rules in a table. When a predefined operation is performed on the data, all the rules in the table are checked to determine if any are invoked by the operation. The method comprises the further steps of, for each rule that is invoked, determining whether the trigger condition of the rule is satisfied; and for each rule having a trigger condition that is satisfied, implementing the process of the rule.

As generally outlined in Figure 3, the preferred embodiment of the invention has the following major elements.

1. Support for definition and evaluation of decision points (these are referred as triggers).

A trigger is a decision point used to gate the invocation of a business rule. There are two types of triggers:

a. Simple Triggers

A simple trigger corresponds to a condition on one column of one row of one table (however the columns may be a quick view, computed, or derived columns as described in

RSW8-2000-0007). This invention provides a custom editor for a Java bean property that allows the column and the data condition to be assigned.

b. Compound Triggers

These are made up of other simple or compound triggers, which are logically combined by AND or OR. Each subtrigger as it as added can be chosen to evaluate to TRUE or FALSE. Any logical structure can be built up this way; the only restriction is that it must be possible to evaluate all the components against the same database table Since quick view, derived and computed columns can be used, this is not restrictive. The custom editor for triggers allows any previously defined triggers to be added as subtriggers to form part of a new compound trigger.

Trigger evaluation is optimized for performance. If a compound trigger is evaluated, the evaluation is suspended as soon as possible (if the subtriggers are combined by AND, evaluation stops as soon as one of them fails; if they are combined by OR, evaluation stops as soon as one passes).

2. Support for creation and implementation of business rules.

A business rule is a verification associated to a single database table. During processing, there are points at which rules for the table are invoked (on add, on change, on update, on delete, on promote). At that point, all

rules for the table that are specified for that event are invoked.

Each rule is gated by a trigger that determines whether to process the rule when it is invoked. This trigger is unique to the rule, but it is created by adding subtriggers that were defined using the trigger creation techniques described above. A list of all valid triggers for the database table is presented for their inclusion as subtriggers.

The creation and definition of rules is done using an editor for a Java bean property. This editor allows selection of a set of triggers for the rule (like compound triggers, the rule and combine subtriggers use AND or OR). The editor also allows the error message and help for the rule to be specified. The events where the rule should be invoked are chosen from within this editor.

The rule may include the name of a class to invoke for further processing. These classes all inherit from a common class. If a Java class for the rule is implemented, its redefinition of the rule processing method is invoked if the rule is triggered during processing (this happens when the events that invoke the rule occur, and the rules trigger gates are passed). These classes are isolated from the rest of the application, which means that the rules can be modified without affecting the rest of the application. The rule processing method gets enough context information so that

it can easily be modified to do any additional processing that may be required.

3. Support for reliable and consistent invocation of the business rules.

The Enterprise Application development platform includes invocation of the business rules at the appropriate points in processing. EADP provides a uniform and consistent application architecture and runtime environment which assures that the rules are properly invoked.

4. Support for both interactive and server side processing.

For interactive processing, EADP provides extensive facilities to flag fields and rows in error, and to display the error message associated to the row. For server side processing, EADP provides a list of message nodes for the error messages, which can be processed to provide all error messages in a batch fashion. Each message node has key information that identifies the data row in error.

5. Isolation of business logic.

This invention totally isolates business logic from any other code in the application.

6. The ability to create business rules without writing Java code.

Trigger and rule definition is done using custom editors. No knowledge of Java or any other programming language is required. It is important to note that this invention does not introduce a specialized scripting language for rule definition.

7. The ability to change business logic without recompiling and reinstalling the application.

Sub A5

The beam to define triggers and rules is implemented as a Dynamic Java Bean (see RSW8-2000-0008). This means that the triggers and rules can be modified in the runtime application, without any need to alter application code. The facility to do this is provided by a Java application that calls up the same special editors used to edit triggers and rules during buildtime customization.

The discussion below describes in greater detail several of the above-identified elements of the preferred embodiment of the invention.

1. Support for definition and evaluation of triggers.

Triggers and rules are both customized using editors that are associated to the custom editor for EADPLogicController (this uses the facilities for dynamic Java beans described in RSW8-2000-0008). The encloser class is a child of EADPApplicationClass (the child class controls a particular database table, for example OrdersApplicationClass would control the Orders table). The bean that is customized is the currentDatamanager of type EADPAManager, and the value in the property sheet of

the bean that is customized is the logicController, of type EADPLogicController. The custom editor for EADPLogicController (named EADPLogicControllerDisplay in conformance with the dynabean standard) has buttons to bring up the custom editors for triggers and rules. The encloser class (e.g. OrdersApplicationClass) is used to provide linkage with the underlying database table (e.g. Orders).

Triggers are defined visually using the EADPTriggerDisplay class. The database and table for the trigger are determined by its encloser class. This class presents a window that allows the designer to define the following:

a. Trigger name

A descriptive name

b. Trigger symbol

A short name without blanks used as the key for the trigger.

c. Trigger description

A longer text to document the business purpose or meaning of the trigger.

d. Data for simple triggers

i. Column

The column in the table that will be evaluated. This is selected from a dropdown list that includes computed or derived columns that have been defined for that table. For EADP Business Factor Tables, this is further refined to use column values in the BFT rather than the raw data value as the source for the comparison.

ii. Operator

A dropdown list of operators to select how the data value will be compared to the trigger value.

iii. Trigger value

The fixed comparison value that will be used to determine if the trigger condition is met.

Sub A6

An example might be a trigger for order cost less that fifty dollars. In this case, the column would be a derived (summary) field that gives the total cost of the order. The comparison operator would be "<" and the trigger value would be 50.

e. Data for compound triggers

i. Subtriggers

A drop down list shows all available triggers (ones that have been added to the logic controller for the encloser class). These may be added to the subtrigger list.

ii. And/Or

This toggle button determines whether AND or OR will be used to evaluate the subtriggers.

iii. True/False

This toggle button is used with a selected subtrigger. It determines if that subtrigger should evaluate to true or false when calculating the combined result.

Existing triggers can also be retrieved and modified (or deleted) from this window.

When a trigger is added, a new instance of EADPTriggerNode is created using the information provided in the window. If the trigger is a compound trigger, the subtriggers are stored in a Vector within the node. are created for each subtrigger, and they are combined in an ordered collection that is stored as an attribute of the trigger. Each entry in the list is a text string that records whether that subtrigger is to be evaluated to true or false, and the symbol for the subtrigger.

The initialization string for the logicController value includes an initialization string that records the trigger definition. The EADPTriggerNode has methods (getString and getJavaString) to create an initialization string for that node. On the reverse side, the node has a setFromString method that takes an appropriate fragment of the initialization string and uses it to set up that node.

Trigger evaluation is provided by the EADPLogicController, which is the logicController property of the EADPDAManager (this is the property customized to set up the triggers). Each trigger node has the current instance of logic controller as an attribute. The logic controller in turn is assigned the current row (as an instance of EADPPeristentObject) when trigger evaluation is required. The trigger node, if it is a simple trigger, uses its column name to find the column value in the current row, and then compares that (using the proper comparison operator) against the trigger value that is part of the trigger node definition. As will be understood by those of ordinary skill in the art, preferably everything is cast into the proper class type for the comparison. If the trigger is a compound trigger, each of the subtriggers is evaluated as described above, and the collection of results is used to determine the result of the trigger. This is done so that the minimum number of subtriggers needs to be evaluated -- if the combining condition is AND, evaluation of the subtriggers continues until one fails. If it is OR, evaluation continues until one succeeds.

2. Support for creation and implementation of business rules.

Like triggers, rules are defined visually by customizing the logicController property. The custom editor for the logicController has a button that brings up the EADPRulesDisplay panel. The database and table for the rules is determined by the encloser (a child of EADPApplicationClass). For example, to define rules for

the Orders table, the logicController property of the data manager bean in OrdersApplicationClass would be customized. Since the rules require triggers to gate the rules, preferably the necessary triggers are defined first.

The EADPRulesDisplay panel presents a window that allows the designer to define the following:

a. Rule name

A descriptive name

b. Rule symbol

A short name without blanks used as the name for the method that will be generated to implement the rule. The symbol is also used as the key for the rule in the Rule Dictionary.

c. Rule description

A longer text to document the business purpose or meaning of the rule.

d. Message text and help

If the rule is not redefined, the default behavior is to issue the error message specified here.

e. Invoke when

This set of toggle buttons is used to determine at what points in processing the rule is to be invoked. The choices are:

On Add
On change
On Update
On Delete
On Promote
On Success

Any combination can be used.

f. Method Class

This is the class name of a class that can be optionally defined to do further processing of the rule. The class needs to inherit from com.ibm.eadp.rules.EADPRuleMethodClass, and redefine the processRule method.

g. Triggers

Each rule has its own trigger, which may be a compound trigger made up of subtriggers defined as described above (the trigger for the rule is stored with the rule and does not show up in the list of triggers defined for the database table).

The processing for defining a compound trigger for the rule is the same as for compound triggers described above.

Rules can also be modified and deleted from this window.

When a rule is saved, a new instance of EADPRuleNode is created in memory. As with the triggers, the rule nodes have methods to generate initialization strings for saving the bean property, and to recreate themselves from the initialization string. The initialization strings for the rule nodes and trigger nodes are combined into the initialization string for the logicController property to achieve its bean customization.

3. Support for reliable and consistent invocation of the business rules.

Rule invocation is provided by the EADPDAManager (see RSW8-2000-0007) and its logicController property (of type EADPLogicController). The logic controller has a ruleController property (of type EADPRuleController). Its processRule method is responsible for doing the rule processing. This method is called by EADPDAManager when any activity (add, change, delete, promote, etc.) is processed against a row (an instance of EADPPersistentObject). The processRule method is passed the type of the update, and the row. It then checks for any rules defined to the logic controller (these are set up using the bean customizations described above). Each rule is checked to see if it is to be invoked for the passed activity type; if so, the passed row is used to evaluate the trigger conditions for the rule. An instance variable (the process result) is used to determine overall success or failure of the repeated calls to processRule during an application process that

involves more than one row. If an error message is issued by rule, an instance of EADPMessageNode is created. The worst return code is tracked; if it is negative, processing stops after the verifications are invoked.

The invocation of rules as events occur ensures that business rules are uniformly enforced. For example, "add" verifications are also invoked on copy.

4. The ability to change business logic without recompiling and reinstalling the application.

The customer editor for the logicController has buttons to bring up the edit panels for rules and triggers. It also has the Register button which it inherits from EADPDynaBeanDisplay (see RSW8-2000-0008). This stores the initialization string for the logicController (which defines the triggers and rules) in a table (the EADPBEAN table) for the database for the encloser. This initialization string can then be updated on the database using EADP dynamic bean facilities.

The preferred embodiment of this invention, as described above, has a number of important advantages.

1. The Enterprise Application Development Platform is unique in that it recognizes a common layer of processing common to all (or at least most) business applications. This allows EADP solutions (including the business logic support presented here) to be applied to a wide variety of businesses.

2. The decomposition of business rules into triggers and rules presented here is an important advantage. This is particularly true for the way compound triggers are defined. This is a simple, efficient and easily maintained way to present complex logical decisions.

3. The ability to update the business logic at runtime using the dynabean facilities is very useful. This offers a comprehensive ability to change any business rule (except the ones implemented by Java methods) without recompiling or reinstalling the application.

4. EADP offers a very optimized application architecture, and the positioning of the business logic provides an excellent level of isolation and very reliable execution.

The present invention has been implemented in the Enterprise Application Development Platform (EADP). The user manual for this facility is included herein a Appendix A.

While it is apparent that the invention herein disclosed is well calculated to fulfill the objects stated above, it will be appreciated that numerous modifications and embodiments may be devised by those skilled in the art, and it is intended that the appended claims cover all such modifications and embodiments as fall within the true spirit and scope of the present invention.